

# TOWARDS QUDA 1.0

Kate Clark, Mathias Wagner and Evan Weinberg

April 27th 2019

PRESENTED BY



# QUDA

- “QCD on CUDA” - <http://lattice.github.com/quda> (open source, BSD license)
- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
  - Pure gauge evolution, gauge fixing, smearing etc.
- Maximize performance
  - Exploit physical symmetries to minimize memory traffic
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Domain-decomposed (Schwarz) preconditioners for strong scaling
  - Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multi-source solvers
  - Multigrid solvers for optimal convergence
- A research tool for how to reach the exascale

# QUDA - LATTICE QCD ON GPUS

<http://lattice.github.com/quda>

 [lattice / quda](#)

Watch ▾ 36    Unstar 45    Fork 29

[Code](#)    [Issues 107](#)    [Pull requests 2](#)    [Wiki](#)    [Pulse](#)    [Graphs](#)    [Settings](#)

QUDA is a library for performing calculations in lattice QCD on GPUs. <http://lattice.github.com/quda> — Edit

4,621 commits    49 branches    19 releases    16 contributors

Branch: [develop](#) ▾    [New pull request](#)    [Create new file](#)    [Upload files](#)    [Find file](#)    [Clone or download](#) ▾

 <a href="#">mathiaswagner</a>	committed on GitHub Merge pull request #487 from lattice/hotfix/checkerboard-reference	...	Latest commit f3e2aa7 a day ago
 <a href="#">include</a>	In ColorSpinorParam, if staggered fermions then set field dimension t...		11 days ago
 <a href="#">lib</a>	Correctly set volumeCB for parity subset references - need to check p...		a day ago
 <a href="#">tests</a>	Requesting --test 1 with staggered_dslash_test now tests MdagM operator		11 days ago
 <a href="#">.gitignore</a>	Updates to .gitignore and renamed multigrid_benchmark to multigrid_be...		3 months ago
 <a href="#">CMakeLists.txt</a>	added some comments to CMakeLists.txt		3 months ago

# QUDA CONTRIBUTORS

Ron Babich (NVIDIA)

Simone Bacchio (Cyprus)

Michael Baldhauf (Regensburg)

Kip Barros (LANL)

Rich Brower (Boston University)

Nuno Cardoso (NCSA)

Kate Clark (NVIDIA)

Michael Cheng (Boston University)

Carleton DeTar (Utah University)

Justin Foley (Utah -> NIH)

Joel Giedt (Rensselaer Polytechnic Institute)

Arjun Gambhir (William and Mary)

Steve Gottlieb (Indiana University)

Kyriakos Hadjiyiannakou (Cyprus)

Dean Howarth (BU)

Bálint Joó (Jlab)

Hyung-Jin Kim (BNL -> Samsung)

Bartek Kostrzewa (Bonn)

Claudio Rebbi (Boston University)

Hauke Sandmeyer (Bielefeld)

Guochun Shi (NCSA -> Google)

Mario Schröck (INFN)

Alexei Strelchenko (FNAL)

Jiqun Tu (Columbia)

Alejandro Vaquero (INFN)

Mathias Wagner (NVIDIA)

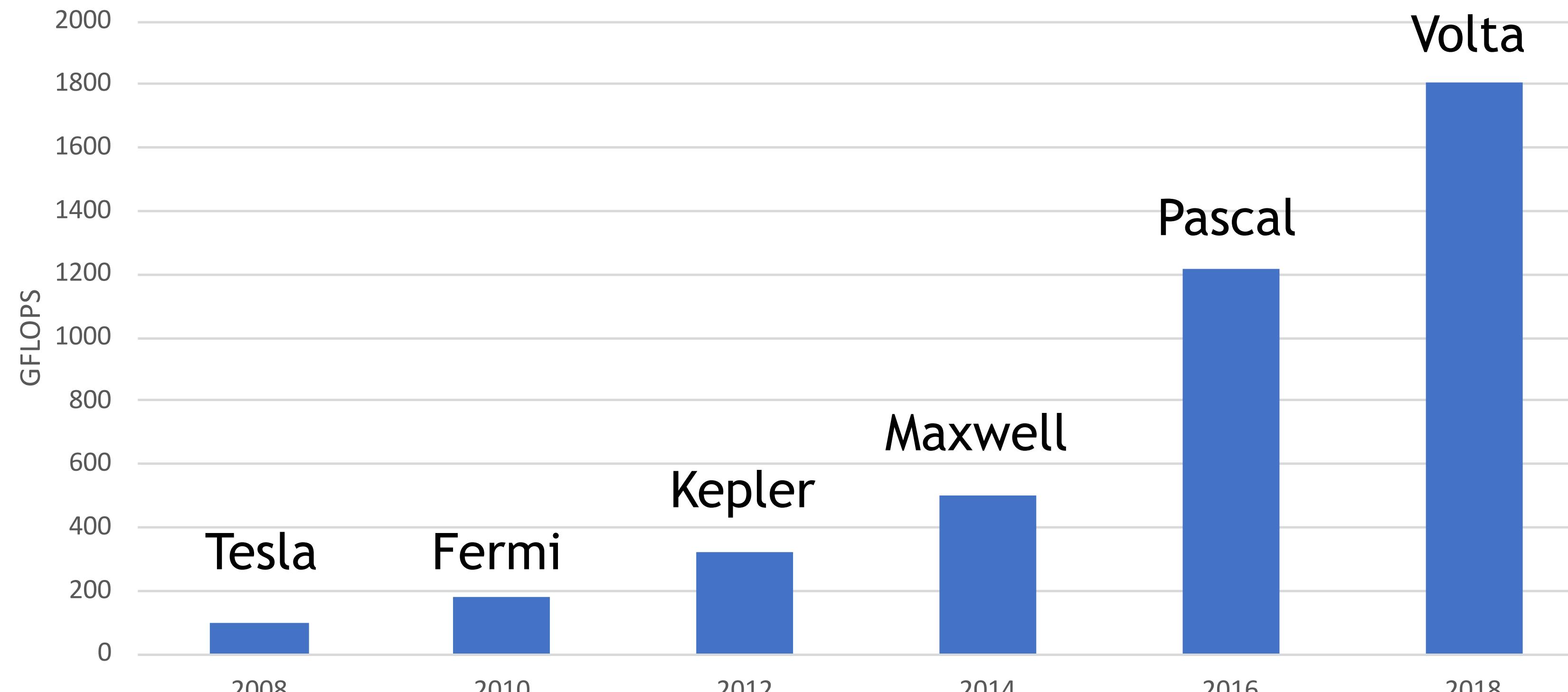
Evan Weinberg (BU -> NVIDIA)

Frank Winter (Jlab)

# DSLASH REWRITE

# WILSON-DSLASH PERFORMANCE

Same code works over 10 years of GPUs



Single precision

```
#define DSLASH_SHARED_FLOATS_PER_THREAD 0
```

```

#ifndef ((CUDA_VERSION >= 4010) && (__COMPUTE_CAPABILITY__ >= 200)) // NVVM compiler
#define VOLATILE
#ifndef __CUDACC__
#define VOLATILE volatile
#endif
// input spinor
#ifndef SPINOR_DOUBLE
#define spinorFloat double
#endif
#define i00_re 10.x
#define i00_im 10.y
#define i01_re 11.x
#define i01_im 11.y
#define i02_re 12.x
#define i02_im 12.y
#define i10_re 13.x
#define i10_im 13.y
#define i11_re 14.x
#define i11_im 14.y
#define i12_re 15.x
#define i12_im 15.y
#define i20_re 16.x
#define i20_im 16.y
#define i21_re 17.x
#define i21_im 17.y
#define i22_re 18.x
#define i22_im 18.y
#define i30_re 19.x
#define i30_im 19.y
#define i31_re 110.x
#define i31_im 110.y
#define i32_re 111.x
#define i32_im 111.y
#define acc00_re accum0.x
#define acc00_im accum0.y
#define g22_im G4.y
#endif // GAUGE_DOUBLE

// conjugated gauge link
#define gT00_re (+g00_re)
#define gT00_im (-g00_im)
#define gT01_re (+g10_re)
#define gT01_im (-g10_im)
#define gT02_re (+g20_re)
#define gT02_im (-g20_im)
#define gT10_re (+g01_re)
#define gT10_im (-g01_im)
#define gT11_re (+g11_re)
#define gT11_im (-g11_im)
#define gT12_re (+g21_re)
#define gT12_im (-g21_im)
#define gT20_re (+g02_re)
#define gT20_im (-g02_im)
#define gT21_re (+g12_re)
#define gT21_im (-g12_im)
#define gT22_re (+g22_re)
#define gT22_im (-g22_im)

// first chiral block of inverted clover term

```

Wonderful  
One eighth  
All old QUD

```
#define i32_re i5.z
#define i32_im i5.w
#define acc0_re accum0.x
#define acc0_im accum0.y
#define acc01_re accum0.z
#define acc01_im accum0.w
#define acc02_re accum1.x
#define acc02_im accum1.y
#define acc10_re accum1.z
#define acc10_im accum1.w
#define acc11_re accum2.x
#define acc11_im accum2.y
#define acc12_re accum2.z
#define acc12_im accum2.w
#define acc20_re accum3.x
#define acc20_im accum3.y
#define acc21_re accum3.z
#define acc21_im accum3.w
#define acc22_re accum4.x
#define acc22_im accum4.y
#define acc30_re accum4.z
#define acc30_im accum4.w
#define acc31_re accum5.x
#define acc31_im accum5.y
#define acc32_re accum5.z
#define acc32_im accum5.w
#endif // SPINOR_DOUBLE
```

# Shouldn't b

```
// gauge link
#ifndef GAUGE_FLOAT2
#define g00_re G0.x
#define g00_im G0.y
#define g01_re G1.x
#define g01_im G1.y
#define g02_re G2.x
#define g02_im G2.y
#define g10_re G3.x
#define g10_im G3.y
#define g11_re G4.x
#define g11_im G4.y
#define g12_re G5.x
#define g12_im G5.y
#define g20_re G6.x
#define g20_im G6.y
#define g21_re G7.x
#define g21_im G7.y
#define g22_re G8.x
#define g22_im G8.y

#else
#define g00_re G0.x
#define g00_im G0.y
#define g01_re G0.z
#define g01_im G0.w
#define g02_re G1.x
#define g02_im G1.y
#define g10_re G1.z
#define g10_im G1.w
#define g11_re G2.x
#define g11_im G2.y
#define g12_re G2.z
#define g12_im G2.w
#define g20_re G3.x
#define g20_im G3.y
#define g21_re G3.z
#define g21_im G3.w
#define g22_re G4.x
```

OLD QU

und is the Python gene

yway...

was this hybrid mess of

ed on your worst enemy

n use of texture cache

(id, param);

```
const int face_volume = (param.threads >> 1);           // volume of one face
const int face_num = (sid >= face_volume);                 // is this thread updating face 0 or 1
face_idx = sid - face_num*face_volume;                      // index into the respective face

// ghostOffset is scaled to include body (includes stride) and number of FloatN arrays (SPINOR_HOP)
// face_idx not sid since faces are spin projected and share the same volume index (modulo UP/DOWN reading)
// /sp_idx = face_idx + param.ghostOffset[dim];

coordsFromFaceIndex<4, QUDA_4D_PC, kernel_type>(X, sid, coord, face_idx, face_num, param);

READ_INTERMEDIATE_SPINOR(INTERTEX, param.sp_stride, sid, sid);

o00_re = i00_re; o00_im = i00_im;
o01_re = i01_re; o01_im = i01_im;
o02_re = i02_re; o02_im = i02_im;
o10_re = i10_re; o10_im = i10_im;
o11_re = i11_re; o11_im = i11_im;
o12_re = i12_re; o12_im = i12_im;
o20_re = i20_re; o20_im = i20_im;
o21_re = i21_re; o21_im = i21_im;
o22_re = i22_re; o22_im = i22_im;
o30_re = i30_re; o30_im = i30_im;
o31_re = i31_re; o31_im = i31_im;
o32_re = i32_re; o32_im = i32_im;
}

#endif // MULTI_GPU

#endif def MULTI_GPU
if ( (kernel_type == INTERIOR_KERNEL && (!param.ghostDim[0] || coord[0]<X1m1)) ||
     (kernel_type == EXTERIOR_KERNEL_X && coord[0]==X1m1) )
#endif
{
    // Projector P0-
    // 1 0 0 -i
    // 0 1 -i 0
    // 0 1 0
    // i 0 0 1

#define MUL__GPU
const int sp_idx = (kernel_type == INTERIOR_KERNEL) ? (coord[0]==Y1m1 ? X+1 : X+1) >> 1 :
    face_idx + param.ghostOffset[static_cast<int>(kernel_type)];
#if (DD_PREC==S) half precision
const int sp_norm_idx = face_idx + param.ghostNormalOffset[static_cast<int>(kernel_type)];
#endif
#else
const int sp_idx = (coord[0]==X1m1 ? X-X1m1 : X+1) >> 1;
#endif

const int ga_idx = sid;

spinorFloat a0_re, a0_im;
spinorFloat a1_re, a1_im;
spinorFloat a2_re, a2_im;
spinorFloat b0_re, b0_im;
spinorFloat b1_re, b1_im;
spinorFloat b2_re, b2_im;

#endif def MULTI_GPU
#endif def kernel_type == INTERIOR_KERNEL {
#endif if

    // read spinor from device memory
    READ_SPINOR(SPINORTEX, param.sp_stride, sp_idx, sp_idx);

    // project spinor into half spinors
    a0_re = +i00_re+i30_im;
    a0_im = +i00_im-i30_re;
    a1_re = +i01_re+i31_im;
    a1_im = +i01_im-i31_re;
    a2_re = +i02_re+i32_im;
    a2_im = +i02_im-i32_re;
    b0_re = +i10_re+i20_im;
    b0_im = +i10_im-i20_re;
    b1_re = +i11_re+i21_im;
    b1_im = +i11_im-i21_re;
    b2_re = +i12_re+i22_im;
    b2_im = +i12_im-i22_re;
}

#endif def MULTI_GPU
else {

    const int sp_stride_pad = ghostFace[static_cast<int>(kernel_type)];

    // read half spinor from device memory
    READ_HALF_SPINOR(GHOSTSPINORTEX, sp_stride_pad, sp_idx, sp_norm_idx);

    a0_re = i00_re; a0_im = i00_im;
    a1_re = i01_re; a1_im = i01_im;
    a2_re = i02_re; a2_im = i02_im;
    b0_re = i10_re; b0_im = i10_im;
    b1_re = i11_re; b1_im = i11_im;
    b2_re = i12_re; b2_im = i12_im;
}

#endif // MULTI_GPU

// read gauge matrix from device memory
READ_GAUGEMATRIX(G, GAUGEOEX, (sp_norm_idx*sp_stride));
// reconstruct gauge matrix
RECONSTRUCT_GAUGE_MATRIX(0);

// multiply row 0
spinorFloat A0_re = 0;
A0_re += g00_re * a0_re;
A0_re -= g00_im * a0_im;
A0_re += g01_re * a1_re;
A0_re -= g01_im * a1_im;
A0_re += g02_re * a2_re;
A0_re -= g02_im * a2_im;
spinorFloat A0_im = 0;
A0_im += g00_re * a0_im;
A0_im += g00_im * a0_re;
A0_im += g01_re * a1_im;
A0_im -= g01_im * a1_re;
A0_im += g02_re * a2_im;
A0_im -= g02_im * a2_re;
spinorFloat B0_re = 0;
B0_re += g00_re * b0_re;
B0_re -= g00_im * b0_im;
B0_re += g01_re * b1_re;
B0_re -= g01_im * b1_im;
B0_re += g02_re * b2_re;
B0_re -= g02_im * b2_im;
spinorFloat B0_im = 0;
B0_im += g00_re * b0_im;
B0_im += g00_im * b0_re;
B0_im += g01_re * b1_im;
B0_im -= g01_im * b1_re;
B0_im += g02_re * b2_im;
B0_im -= g02_im * b2_re;

// multiply row 1
spinorFloat A1_re = 0;
A1_re += g10_re * a0_re;
A1_re -= g10_im * a0_im;
A1_re += g11_re * a1_re;
A1_re -= g11_im * a1_im;
A1_re += g12_re * a2_re;
A1_re -= g12_im * a2_im;
spinorFloat A1_im = 0;
A1_im += g10_re * a0_im;
A1_im += g10_im * a0_re;
A1_im += g11_re * a1_im;
A1_im -= g11_im * a1_re;
A1_im += g12_re * a2_im;
A1_im -= g12_im * a2_re;

// multiply row 2
spinorFloat A2_re = 0;
A2_re += g20_re * a0_re;
A2_re -= g20_im * a0_im;
A2_re += g21_re * a1_re;
A2_re -= g21_im * a1_im;
A2_re += g22_re * a2_re;
A2_re -= g22_im * a2_im;
spinorFloat A2_im = 0;
A2_im += g20_re * a0_im;
A2_im += g20_im * a0_re;
A2_im += g21_re * a1_im;
A2_im -= g21_im * a1_re;
A2_im += g22_re * a2_im;
A2_im -= g22_im * a2_re;

// multiply row 3
spinorFloat B1_re = 0;
B1_re += g10_re * b0_re;
B1_re -= g10_im * b0_im;
B1_re += g11_re * b1_re;
B1_re -= g11_im * b1_im;
B1_re += g12_re * b2_re;
B1_re -= g12_im * b2_im;
spinorFloat B1_im = 0;
B1_im += g10_re * b0_im;
B1_im += g10_im * b0_re;
B1_im += g11_re * b1_im;
B1_im -= g11_im * b1_re;
B1_im += g12_re * b2_im;
B1_im -= g12_im * b2_re;

// multiply row 4
spinorFloat A3_re = 0;
A3_re += g20_re * a0_re;
A3_re -= g20_im * a0_im;
A3_re += g21_re * a1_re;
A3_re -= g21_im * a1_im;
A3_re += g22_re * a2_re;
A3_re -= g22_im * a2_im;
spinorFloat A3_im = 0;
A3_im += g20_re * a0_im;
A3_im += g20_im * a0_re;
A3_im += g21_re * a1_im;
A3_im -= g21_im * a1_re;
A3_im += g22_re * a2_im;
A3_im -= g22_im * a2_re;

// multiply row 5
spinorFloat B2_re = 0;
B2_re += g20_re * b0_re;
B2_re -= g20_im * b0_im;
B2_re += g21_re * b1_re;
B2_re -= g21_im * b1_im;
B2_re += g22_re * b2_re;
B2_re -= g22_im * b2_im;
spinorFloat B2_im = 0;
B2_im += g20_re * b0_im;
B2_im += g20_im * b0_re;
B2_im += g21_re * b1_im;
B2_im -= g21_im * b1_re;
B2_im += g22_re * b2_im;
B2_im -= g22_im * b2_re;

// multiply row 6
spinorFloat A4_re = 0;
A4_re += g10_re * a0_re;
A4_re -= g10_im * a0_im;
A4_re += g11_re * a1_re;
A4_re -= g11_im * a1_im;
A4_re += g12_re * a2_re;
A4_re -= g12_im * a2_im;
spinorFloat A4_im = 0;
A4_im += g10_re * a0_im;
A4_im += g10_im * a0_re;
A4_im += g11_re * a1_im;
A4_im -= g11_im * a1_re;
A4_im += g12_re * a2_im;
A4_im -= g12_im * a2_re;

// multiply row 7
spinorFloat B4_re = 0;
B4_re += g20_re * b0_re;
B4_re -= g20_im * b0_im;
B4_re += g21_re * b1_re;
B4_re -= g21_im * b1_im;
B4_re += g22_re * b2_re;
B4_re -= g22_im * b2_im;
spinorFloat B4_im = 0;
B4_im += g20_re * b0_im;
B4_im += g20_im * b0_re;
B4_im += g21_re * b1_im;
B4_im -= g21_im * b1_re;
B4_im += g22_re * b2_im;
B4_im -= g22_im * b2_re;

// multiply row 8
spinorFloat A5_re = 0;
A5_re += g10_re * a0_re;
A5_re -= g10_im * a0_im;
A5_re += g11_re * a1_re;
A5_re -= g11_im * a1_im;
A5_re += g12_re * a2_re;
A5_re -= g12_im * a2_im;
spinorFloat A5_im = 0;
A5_im += g10_re * a0_im;
A5_im += g10_im * a0_re;
A5_im += g11_re * a1_im;
A5_im -= g11_im * a1_re;
A5_im += g12_re * a2_im;
A5_im -= g12_im * a2_re;

// multiply row 9
spinorFloat B5_re = 0;
B5_re += g20_re * b0_re;
B5_re -= g20_im * b0_im;
B5_re += g21_re * b1_re;
B5_re -= g21_im * b1_im;
B5_re += g22_re * b2_re;
B5_re -= g22_im * b2_im;
spinorFloat B5_im = 0;
B5_im += g20_re * b0_im;
B5_im += g20_im * b0_re;
B5_im += g21_re * b1_im;
B5_im -= g21_im * b1_re;
B5_im += g22_re * b2_im;
B5_im -= g22_im * b2_re;

// multiply row 10
spinorFloat A6_re = 0;
A6_re += g10_re * a0_re;
A6_re -= g10_im * a0_im;
A6_re += g11_re * a1_re;
A6_re -= g11_im * a1_im;
A6_re += g12_re * a2_re;
A6_re -= g12_im * a2_im;
spinorFloat A6_im = 0;
A6_im += g10_re * a0_im;
A6_im += g10_im * a0_re;
A6_im += g11_re * a1_im;
A6_im -= g11_im * a1_re;
A6_im += g12_re * a2_im;
A6_im -= g12_im * a2_re;

// multiply row 11
spinorFloat B6_re = 0;
B6_re += g20_re * b0_re;
B6_re -= g20_im * b0_im;
B6_re += g21_re * b1_re;
B6_re -= g21_im * b1_im;
B6_re += g22_re * b2_re;
B6_re -= g22_im * b2_im;
spinorFloat B6_im = 0;
B6_im += g20_re * b0_im;
B6_im += g20_im * b0_re;
B6_im += g21_re * b1_im;
B6_im -= g21_im * b1_re;
B6_im += g22_re * b2_im;
B6_im -= g22_im * b2_re;

// multiply row 12
spinorFloat A7_re = 0;
A7_re += g10_re * a0_re;
A7_re -= g10_im * a0_im;
A7_re += g11_re * a1_re;
A7_re -= g11_im * a1_im;
A7_re += g12_re * a2_re;
A7_re -= g12_im * a2_im;
spinorFloat A7_im = 0;
A7_im += g10_re * a0_im;
A7_im += g10_im * a0_re;
A7_im += g11_re * a1_im;
A7_im -= g11_im * a1_re;
A7_im += g12_re * a2_im;
A7_im -= g12_im * a2_re;

// multiply row 13
spinorFloat B7_re = 0;
B7_re += g20_re * b0_re;
B7_re -= g20_im * b0_im;
B7_re += g21_re * b1_re;
B7_re -= g21_im * b1_im;
B7_re += g22_re * b2_re;
B7_re -= g22_im * b2_im;
spinorFloat B7_im = 0;
B7_im += g20_re * b0_im;
B7_im += g20_im * b0_re;
B7_im += g21_re * b1_im;
B7_im -= g21_im * b1_re;
B7_im += g22_re * b2_im;
B7_im -= g22_im * b2_re;

// multiply row 14
spinorFloat A8_re = 0;
A8_re += g10_re * a0_re;
A8_re -= g10_im * a0_im;
A8_re += g11_re * a1_re;
A8_re -= g11_im * a1_im;
A8_re += g12_re * a2_re;
A8_re -= g12_im * a2_im;
spinorFloat A8_im = 0;
A8_im += g10_re * a0_im;
A8_im += g10_im * a0_re;
A8_im += g11_re * a1_im;
A8_im -= g11_im * a1_re;
A8_im += g12_re * a2_im;
A8_im -= g12_im * a2_re;

// multiply row 15
spinorFloat B8_re = 0;
B8_re += g20_re * b0_re;
B8_re -= g20_im * b0_im;
B8_re += g21_re * b1_re;
B8_re -= g21_im * b1_im;
B8_re += g22_re * b2_re;
B8_re -= g22_im * b2_im;
spinorFloat B8_im = 0;
B8_im += g20_re * b0_im;
B8_im += g20_im * b0_re;
B8_im += g21_re * b1_im;
B8_im -= g21_im * b1_re;
B8_im += g22_re * b2_im;
B8_im -= g22_im * b2_re;

// multiply row 16
spinorFloat A9_re = 0;
A9_re += g10_re * a0_re;
A9_re -= g10_im * a0_im;
A9_re += g11_re * a1_re;
A9_re -= g11_im * a1_im;
A9_re += g12_re * a2_re;
A9_re -= g12_im * a2_im;
spinorFloat A9_im = 0;
A9_im += g10_re * a0_im;
A9_im += g10_im * a0_re;
A9_im += g11_re * a1_im;
A9_im -= g11_im * a1_re;
A9_im += g12_re * a2_im;
A9_im -= g12_im * a2_re;

// multiply row 17
spinorFloat B9_re = 0;
B9_re += g20_re * b0_re;
B9_re -= g20_im * b0_im;
B9_re += g21_re * b1_re;
B9_re -= g21_im * b1_im;
B9_re += g22_re * b2_re;
B9_re -= g22_im * b2_im;
spinorFloat B9_im = 0;
B9_im += g20_re * b0_im;
B9_im += g20_im * b0_re;
B9_im += g21_re * b1_im;
B9_im -= g21_im * b1_re;
B9_im += g22_re * b2_im;
B9_im -= g22_im * b2_re;

// multiply row 18
spinorFloat A10_re = 0;
A10_re += g10_re * a0_re;
A10_re -= g10_im * a0_im;
A10_re += g11_re * a1_re;
A10_re -= g11_im * a1_im;
A10_re += g12_re * a2_re;
A10_re -= g12_im * a2_im;
spinorFloat A10_im = 0;
A10_im += g10_re * a0_im;
A10_im += g10_im * a0_re;
A10_im += g11_re * a1_im;
A10_im -= g11_im * a1_re;
A10_im += g12_re * a2_im;
A10_im -= g12_im * a2_re;

// multiply row 19
spinorFloat B10_re = 0;
B10_re += g20_re * b0_re;
B10_re -= g20_im * b0_im;
B10_re += g21_re * b1_re;
B10_re -= g21_im * b1_im;
B10_re += g22_re * b2_re;
B10_re -= g22_im * b2_im;
spinorFloat B10_im = 0;
B10_im += g20_re * b0_im;
B10_im += g20_im * b0_re;
B10_im += g21_re * b1_im;
B10_im -= g21_im * b1_re;
B10_im += g22_re * b2_im;
B10_im -= g22_im * b2_re;

// multiply row 20
spinorFloat A11_re = 0;
A11_re += g10_re * a0_re;
A11_re -= g10_im * a0_im;
A11_re += g11_re * a1_re;
A11_re -= g11_im * a1_im;
A11_re += g12_re * a2_re;
A11_re -= g12_im * a2_im;
spinorFloat A11_im = 0;
A11_im += g10_re * a0_im;
A11_im += g10_im * a0_re;
A11_im += g11_re * a1_im;
A11_im -= g11_im * a1_re;
A11_im += g12_re * a2_im;
A11_im -= g12_im * a2_re;

// multiply row 21
spinorFloat B11_re = 0;
B11_re += g20_re * b0_re;
B11_re -= g20_im * b0_im;
B11_re += g21_re * b1_re;
B11_re -= g21_im * b1_im;
B11_re += g22_re * b2_re;
B11_re -= g22_im * b2_im;
spinorFloat B11_im = 0;
B11_im += g20_re * b0_im;
B11_im += g20_im * b0_re;
B11_im += g21_re * b1_im;
B11_im -= g21_im * b1_re;
B11_im += g22_re * b2_im;
B11_im -= g22_im * b2_re;

// multiply row 22
spinorFloat A12_re = 0;
A12_re += g10_re * a0_re;
A12_re -= g10_im * a0_im;
A12_re += g11_re * a1_re;
A12_re -= g11_im * a1_im;
A12_re += g12_re * a2_re;
A12_re -= g12_im * a2_im;
spinorFloat A12_im = 0;
A12_im += g10_re * a0_im;
A12_im += g10_im * a0_re;
A12_im += g11_re * a1_im;
A12_im -= g11_im * a1_re;
A12_im += g12_re * a2_im;
A12_im -= g12_im * a2_re;

// multiply row 23
spinorFloat B12_re = 0;
B12_re += g20_re * b0_re;
B12_re -= g20_im * b0_im;
B12_re += g21_re * b1_re;
B12_re -= g21_im * b1_im;
B12_re += g22_re * b2_re;
B12_re -= g22_im * b2_im;
spinorFloat B12_im = 0;
B12_im += g20_re * b0_im;
B12_im += g20_im * b0_re;
B12_im += g21_re * b1_im;
B12_im -= g21_im * b1_re;
B12_im += g22_re * b2_im;
B12_im -= g22_im * b2_re;
```

# OLD QUDA KERNELS

# Wonderful background is the Python generated Wilsonslash kernel

One eighth of it any

All old QUDA code was this hybrid mess of Python and / or C macros

Shouldn't be inflicted on your worst enemy

Performance rely on use of texture cache

```

#define c00_10_im (<i0_00_im>)
#define c01_10_re (+c10_01_re)
#define c01_10_im (<c10_01_im>)
#define c02_10_re (+c10_02_re)
#define c02_10_im (<i0_02_im>)
#define c00_11_re (+c11_00_re)
#define c00_11_im (<i11_00_im>)
#define c01_11_re (+c11_01_re)
#define c01_11_im (<c11_01_im>)
#define c02_11_re (+c11_02_re)
#define c02_11_im (<i11_02_im>)
#define c10_11_re (+c11_10_re)
#define c10_11_im (<i11_10_im>)
#define c00_12_re (+c12_00_re)
#define c00_12_im (<i12_00_im>)
#define c01_12_re (+c12_01_re)
#define c01_12_im (<c12_01_im>)
#define c02_12_re (+c12_02_re)
#define c02_12_im (<i12_02_im>)
#define c10_12_re (+c12_10_re)
#define c10_12_im (<i12_10_im>)
#define c11_12_re (+c12_11_re)
#define c11_12_im (<c12_11_im>)

#ifndef MULTILGPU
    #define blockIdx.x blockIdx.x*blockDim.x + threadIdx.x;
    #define if (sid >= param.threads) return;
    #define coordsFromIndex<4,QUDA_4D_PC,EVEN_X>(X, coord, sid, param);
    #define o00_re = 0; o00_im = 0;
    #define o01_re = 0; o01_im = 0;
    #define o02_re = 0; o02_im = 0;
    #define o10_re = 0; o10_im = 0;
    #define o11_re = 0; o11_im = 0;
    #define o12_re = 0; o12_im = 0;
    #define o20_re = 0; o20_im = 0;
    #define o21_re = 0; o21_im = 0;
    #define o22_re = 0; o22_im = 0;
    #define o30_re = 0; o30_im = 0;
    #define o31_re = 0; o31_im = 0;
    #define o32_re = 0; o32_im = 0;
    #if !defined MULTILGPU
        } else { // exterior kernel
            if (sid >= param.threads) return;
            int sid = blockIdx.x*blockDim.x + threadIdx.x;
            if (sid >= param.threads) return;
            coordsFromIndex<4,QUDA_4D_PC,EVEN_X>(X, coord, sid, param);
            o00_re = g00_re * b0_re;
            o00_im = g00_im * b0_im;
            o01_re += g01_re * b1_re;
            o01_im += g01_im * b1_im;
            o02_re += g02_re * b2_re;
            o02_im += g02_im * b2_im;
            spinorFloat B0_im = 0;
            B0_im += g00_re * b0_im;
            B0_im += g00_im * b0_re;
            B0_im += g01_re * b1_im;
            B0_im += g01_im * b1_re;
            B0_im += g02_re * b2_im;
            B0_im += g02_im * b2_re;
            // multiply row 1
            spinorFloat A1_re = 0;
            A1_re += g10_re * a0_re;
            A1_re -= g10_im * a0_im;
            A1_re += g11_re * a1_re;
            A1_re -= g11_im * a1_im;
            A1_re += g12_re * a2_re;
            A1_re -= g12_im * a2_im;
            spinorFloat A1_im = 0;
            A1_im += g10_re * a0_im;
            A1_im += g10_im * a0_re;
            A1_im += g11_re * a1_im;
            A1_im += g11_im * a1_re;
            A1_im += g12_re * a2_im;
            A1_im += g12_im * a2_re;
        }
    #endif // MULTILGPU

```

# THE NEED FOR A COMPLETE REWRITE

Extensibility, composability and maintainability

Ability to add new discretizations easily

Integration with `jitify`

Changing representation,  $N_c$  etc.

Ability to run on CPU?

# DSLASH FRAMEWORK

All Dslash kernels now completely rewritten new C++ framework

Reuse the same common cores for multiple stencils

All stencils now support overlapping optimized overlapping comms and compute

Easy to add support for new stencils, change  $N_c$ , representation, etc.

Merged into QUDA develop branch a <https://github.com/lattice/quda/pull/776>

<i>Wilson</i>	✓
<i>Wilson-clover</i>	✓
<i>Twisted-mass singlet</i>	✓
<i>Twisted-mass doublet</i>	✓
<i>Twisted-clover singlet</i>	✓
<i>Naive staggered</i>	✓
<i>Improved staggered</i>	✓
<i>Shamir 5-d</i>	✓
<i>Shamir 4-d</i>	✓
<i>Möbius</i>	✓
<i>Laplace</i>	✓
<i>Laplace-3d</i>	✓
<i>Covariant derivative</i>	✓
<i>Staggered sextet</i>	WIP
<i>Twisted-clover doublet</i>	WIP
<i>Clover “Hasenbusch”</i>	WIP
<i>Block MDWF preconditioner</i>	WIP

```

template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType kernel_type,
typename Arg, typename Vector>
__device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int coord[nDim], int x_cb, int s,
                                         int parity, int idx, int thread_dim, bool &active) {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,2> HalfVector;
    typedef Matrix<complex<real>,nColor> Link;
    const int their_spinor_parity = nParity == 2 ? 1-parity : 0;

#pragma unroll
for (int d = 0; d<nDim; d++) { // loop over dimension
{ // Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = getNeighborIndexCB(coord, d, +1, arg.dc);
    constexpr int proj_dir = dagger ? +1 : -1;
    const bool ghost = (coord[d] + arg.nFace >= arg.dim[d]) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

    if ( doHalo<kernel_type>(d) && ghost ) {
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace) : idx;

        Link U = arg.U(d, x_cb, parity);
        HalfVector in = arg.in.Ghost(d, 1, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale;
        out += (U * in).reconstruct(d, proj_dir);
    } else if ( doBulk<kernel_type>() && !ghost ) {

        Link U = arg.U(d, x_cb, parity);
        Vector in = arg.in(fwd_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

        out += (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
    }
}

{ // Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = getNeighborIndexCB(coord, d, -1, arg.dc);
    const int gauge_idx = back_idx;
    constexpr int proj_dir = dagger ? -1 : +1;
    const bool ghost = (coord[d] - arg.nFace < 0) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

    if ( doHalo<kernel_type>(d) && ghost ) {
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace) : idx;

        Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
        HalfVector in = arg.in.Ghost(d, 0, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale;

        out += (conj(U) * in).reconstruct(d, proj_dir);
    } else if ( doBulk<kernel_type>() && !ghost ) {

        Link U = arg.U(d, gauge_idx, 1-parity);
        Vector in = arg.in(back_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

        out += (conj(U) * in.project(d, proj_dir)).reconstruct(d, proj_dir);
    }
}
} //nDim
}

```

```

//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
__device__ __host__ inline void wilson(Arg &arg, int idx, int parity)
{
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true; // is thread active (non-trival for
fused kernel only)
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;
    Vector out;
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity, idx,
thread_dim, active);

    if (xpay && kernel_type == INTERIOR_KERNEL) {
        Vector x = arg.x(x_cb, my_spinor_parity);
        out = x + arg.kappa * out;
    } else if (kernel_type != INTERIOR_KERNEL && active) {
        Vector x = arg.out(x_cb, my_spinor_parity);
        out = x + (xpay ? arg.kappa * out : out);
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
}

// CPU kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
void wilsonCPU(Arg arg)
{
    for (int parity= 0; parity < nParity; parity++) {
        // for full fields then set parity from loop else use arg setting
        parity = nParity == 2 ? parity : arg.parity;

        for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
            wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, parity);
        } // 4-d volumeCB
    } // parity
}

// GPU Kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
__global__ void wilsonGPU(Arg arg)
{
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 0); break;
    case 1: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 1); break;
    }
}

```

# WILSON KERNEL

```

template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType kernel_type,
typename Arg, typename Vector>
__device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int coord[nDim], int x_cb, int s,
                                         int parity, int idx, int thread_dim, bool &active) {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,2> HalfVector;
    typedef Matrix<complex<real>,nColor> Link;
    const int their_spinor_parity = nParity == 2 ? 1-parity : 0;

#pragma unroll
for (int d = 0; d<nDim; d++) { // loop over dimension
{ // Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = getNeighborIndexCB(coord, d, +1, arg.dc);
    constexpr int proj_dir = dagger ? +1 : -1;
    const bool ghost = (coord[d] + arg.nFace >= arg.dim[d]) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

    if (doHalo<kernel_type>(d) && ghost) {
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<1>(coord, arg.dim, d, arg.nFace) : idx;

        Link U = arg.U(d, x_cb, parity);
        HalfVector in = arg.in.Ghost(d, 1, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale;
        out += (U * in).reconstruct(d, proj_dir);
    } else if (doBulk<kernel_type>() && !ghost) {
        Link U = arg.U(d, x_cb, parity);
        Vector in = arg.in(fwd_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);
        out += (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
    }
}

{ // Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = getNeighborIndexCB(coord, d, -1, arg.dc);
    const int gauge_idx = back_idx;
    constexpr int proj_dir = dagger ? -1 : +1;
    const bool ghost = (coord[d] - arg.nFace < 0) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

    if (doHalo<kernel_type>(d) && ghost) {
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<0>(coord, arg.dim, d, arg.nFace) : idx;

        Link U = arg.U.Ghost(d, ghost_idx, 1-parity);
        HalfVector in = arg.in.Ghost(d, 0, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale;

        out += (conj(U) * in).reconstruct(d, proj_dir);
    } else if (doBulk<kernel_type>() && !ghost) {
        Link U = arg.U(d, gauge_idx, 1-parity);
        Vector in = arg.in(back_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);
        out += (conj(U) * in.project(d, proj_dir)).reconstruct(d, proj_dir);
    }
}
//nDim
}

```

Halo

Interior

Halo

Interior

# WILSON KERNEL

```

//out(x) = M*in = (-D + m) * in(x-mu)
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
__device__ __host__ inline void wilson(Arg &arg, int idx, int parity)
{
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true; // is thread active (non-trivial for
fused kernel only)
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;
    Vector out;
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity, idx,
thread_dim, active);

    if (xpay && kernel_type == INTERIOR_KERNEL) {
        Vector x = arg.x(x_cb, my_spinor_parity);
        out = x + arg.kappa * out;
    } else if (kernel_type != INTERIOR_KERNEL && active) {
        Vector x = arg.out(x_cb, my_spinor_parity);
        out = x + (xpay ? arg.kappa * out : out);
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
}

// CPU kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
void wilsonCPU(Arg arg)
{
    for (int parity= 0; parity < nParity; parity++) {
        // for full fields then set parity from loop else use arg setting
        parity = nParity == 2 ? parity : arg.parity;

        for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
            wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, parity);
        } // 4-d volumeCB
    } // parity
}

// GPU Kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
_global_ void wilsonGPU(Arg arg)
{
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 0); break;
    case 1: wilson<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 1); break;
    }
}

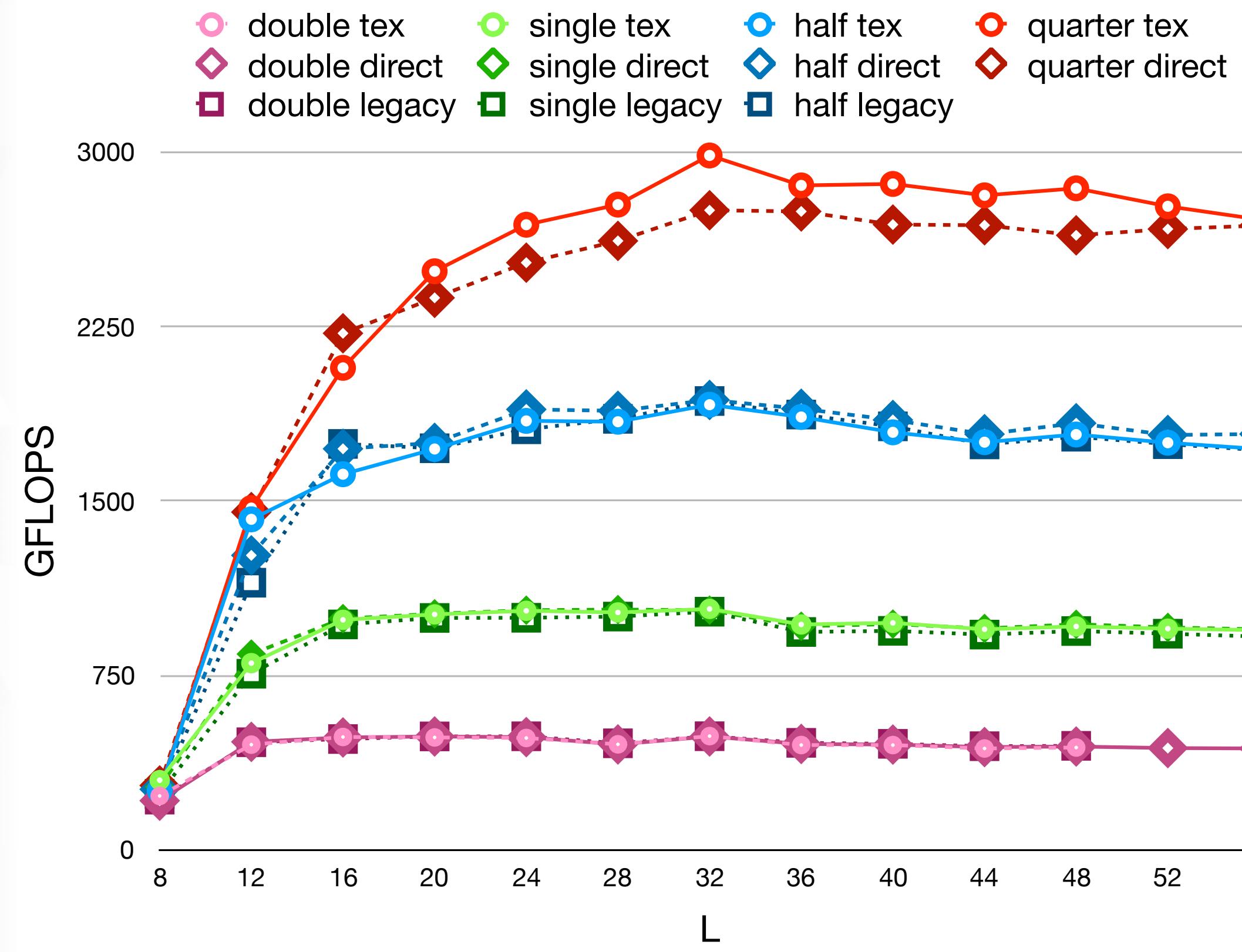
```

CPU

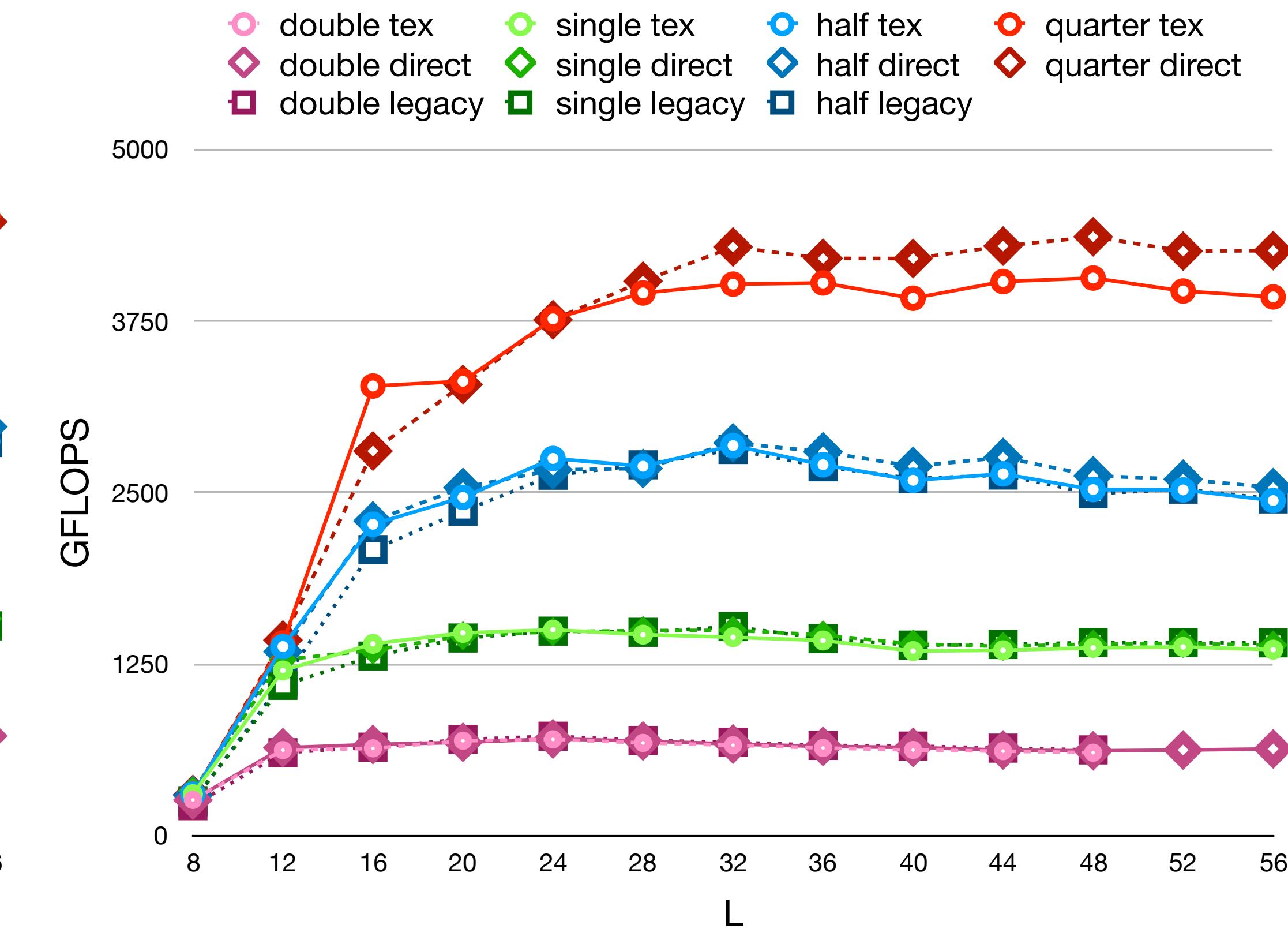
GPU

# PERFORMANCE

Legacy ~ 40K LOC total  
New ~ 1500 LOC



Pascal



Volta

```

template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType kernel_type, typename Arg, typename Vector>
__device__ __host__ inline void applyWilson(Vector &out, Arg &arg, int coord[nDim], int x_cb, int s,
                                         int parity, int idx, int thread_dim, bool &active) {
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,2> HalfVector;
    typedef Matrix<complex<real>,nColor> Link;
    const int their_spinor_parity = nParity == 2 ? 1-parity : 0;

    // parity for gauge field - include residual parity from 5-d => 4-d checkerboarding
    const int gauge_parity = (nDim == 5 ? (x_cb/arg.dc.volume_4d_cb + parity) % 2 : parity);

#pragma unroll
for (int d = 0; d<4; d++) { // loop over dimension
    // Forward gather - compute fwd offset for vector fetch
    const int fwd_idx = getNeighborIndexCB<nDim>(coord, d, +1, arg.dc);
    const int gauge_idx = (nDim == 5 ? x_cb % arg.dc.volume_4d_cb : x_cb);
    constexpr int proj_dir = dagger ? +1 : -1;

    const bool ghost = (coord[d] + arg.nFace >= arg.dim[d]) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

    if ( doHalo<kernel_type>(d) && ghost ) {
        // we need to compute the face index if we are updating a face that isn't ours
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<1,nDim>(coord, arg.dim, d, arg.nFace) : idx;

        Link U = arg.U(d, gauge_idx, gauge_parity);
        HalfVector in = arg.in.Ghost(d, 1, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale; // put this in the Ghost accessor and merge with any rescaling?

        out += (U * in).reconstruct(d, proj_dir);
    } else if ( doBulk<kernel_type>() && !ghost ) {

        Link U = arg.U(d, gauge_idx, gauge_parity);
        Vector in = arg.in(fwd_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

        out += (U * in.project(d, proj_dir)).reconstruct(d, proj_dir);
    }
}

{ // Backward gather - compute back offset for spinor and gauge fetch
    const int back_idx = getNeighborIndexCB<nDim>(coord, d, -1, arg.dc);
    const int gauge_idx = (nDim == 5 ? back_idx % arg.dc.volume_4d_cb : back_idx);
    constexpr int proj_dir = dagger ? -1 : +1;

    const bool ghost = (coord[d] - arg.nFace < 0) &&
        isActive<kernel_type>(active, thread_dim, d, coord, arg);

    if ( doHalo<kernel_type>(d) && ghost ) {
        // we need to compute the face index if we are updating a face that isn't ours
        const int ghost_idx = (kernel_type == EXTERIOR_KERNEL_ALL && d != thread_dim) ?
            ghostFaceIndex<0,nDim>(coord, arg.dim, d, arg.nFace) : idx;

        const int gauge_ghost_idx = (nDim == 5 ? ghost_idx % arg.dc.ghostFaceCB[d] : ghost_idx);
        Link U = arg.U.Ghost(d, gauge_ghost_idx, 1-gauge_parity);
        HalfVector in = arg.in.Ghost(d, 0, ghost_idx+s*arg.dc.ghostFaceCB[d], their_spinor_parity);
        if (d == 3) in *= arg.t_proj_scale;

        out += (conj(U) * in).reconstruct(d, proj_dir);
    } else if ( doBulk<kernel_type>() && !ghost ) {

        Link U = arg.U(d, gauge_idx, 1-gauge_parity);
        Vector in = arg.in(back_idx+s*arg.dc.volume_4d_cb, their_spinor_parity);

        out += (conj(U) * in.project(d, proj_dir)).reconstruct(d, proj_dir);
    }
}
} //nDim
}

```

# COMPOSABILITY

Same Wilson dslash kernel is used by all Wilson-like operators

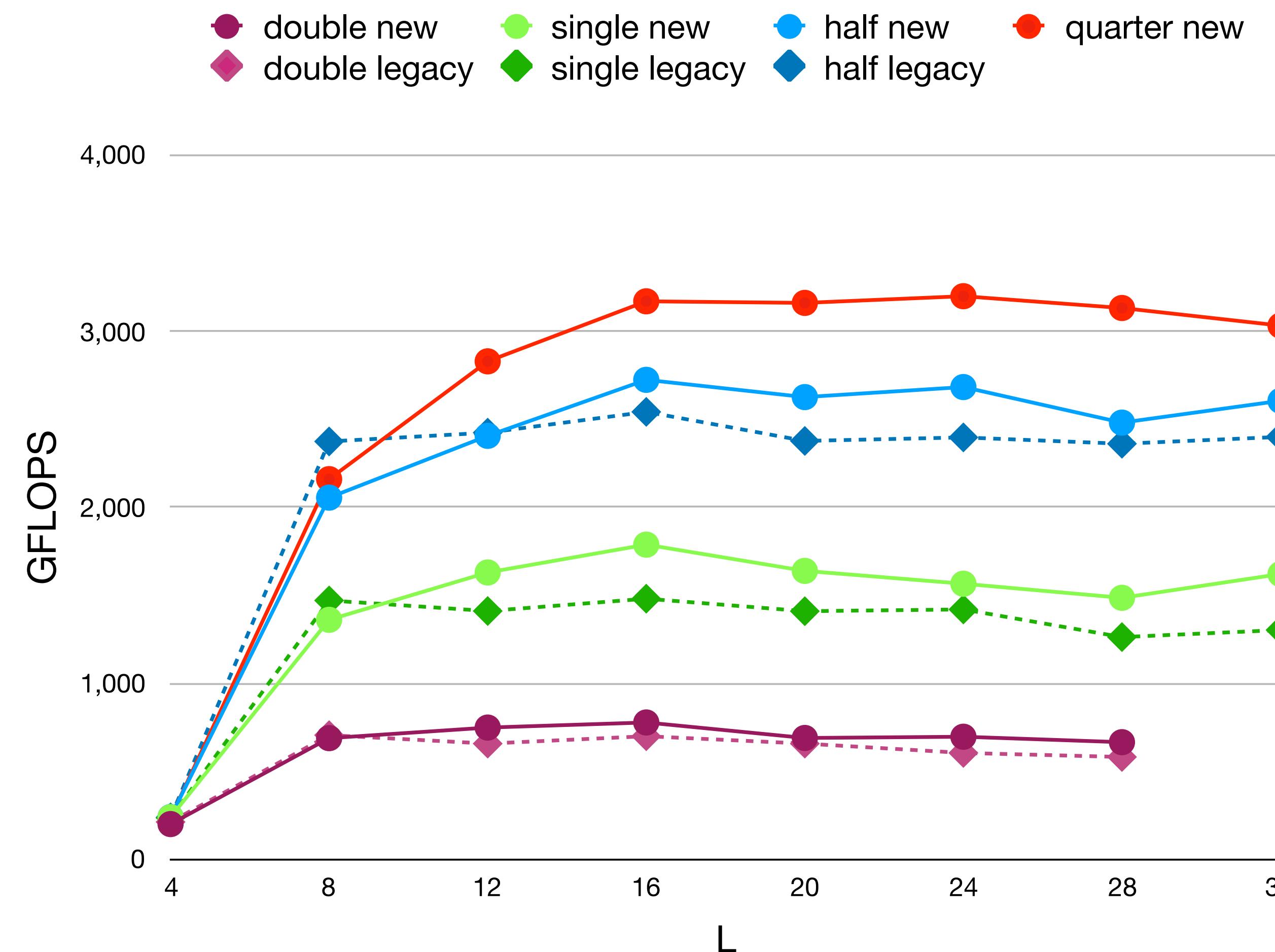
- Wilson
- Clover
- Twisted mass
- Twisted clover
- Shamir 4-d/5-d
- Möbius

Similar for staggered kernel

- Naive
- Improved
- Sextet fermions

# 4D-PRECONDITIONED SHAMIR

Quadro P100



*Sierra CG MDWF Performance  
Volume = 48<sup>3</sup>x64x12  
(Courtesy of CalLat)*

Nodes	Performance	Perf/GPU
4[old]	27300	1706
4	35615	2226
3	29100	2425
2	25312	3164

# EMBRACING NEW IDEAS

Prior framework meant embracing new algorithms and technologies was very time consuming, requiring lots of hand editing, or simply not tractable

<i>Algorithms</i>	<i>Technologies</i>
<i>Multi-rhs solvers</i>	<i>SHMEM</i>
<i>New precisions (quarter and quad?)</i>	<i>Large memory</i>
<i>Schwarz preconditioners</i>	<i>CUDA graphs</i>
<i>Cache and register tiling</i>	
<i>Finer-grain parallelism</i>	

# JITIFY INTEGRATION

## Framework supports just-in-time compilation

```
#include <kernels/dslash_wilson.cuh>
namespace quda {

    template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType kernel_type, typename Arg>
    struct WilsonLaunch {
        static constexpr const char *kernel = "quda::wilsonGPU"; // kernel name for jit compilation
        template <typename Dslash>
        inline static void launch(Dslash &dslash, TuneParam &tp, Arg &arg, const cudaStream_t &stream) {
            dslash.launch(wilsonGPU<Float,nDim,nColor,nParity,dagger,xpay,kernel_type,Arg>, tp, arg, stream);
        }
    };

    template <typename Float, int nDim, int nColor, typename Arg>
    class Wilson : public Dslash<Float> {

protected:
    Arg &arg;
    const ColorSpinorField &in;

public:
    Wilson(Arg &arg, const ColorSpinorField &out, const ColorSpinorField &in)
        : Dslash<Float>(arg, out, in, "kernels/dslash_wilson.cuh"), arg(arg), in(in) { }

    virtual ~Wilson() { }

    void apply(const cudaStream_t &stream) {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        Dslash<Float>::setParam(arg);
        Dslash<Float>::template instantiate<WilsonLaunch,nDim,nColor>(tp, arg, stream);
    }
};
```

Offline: kernel is an include

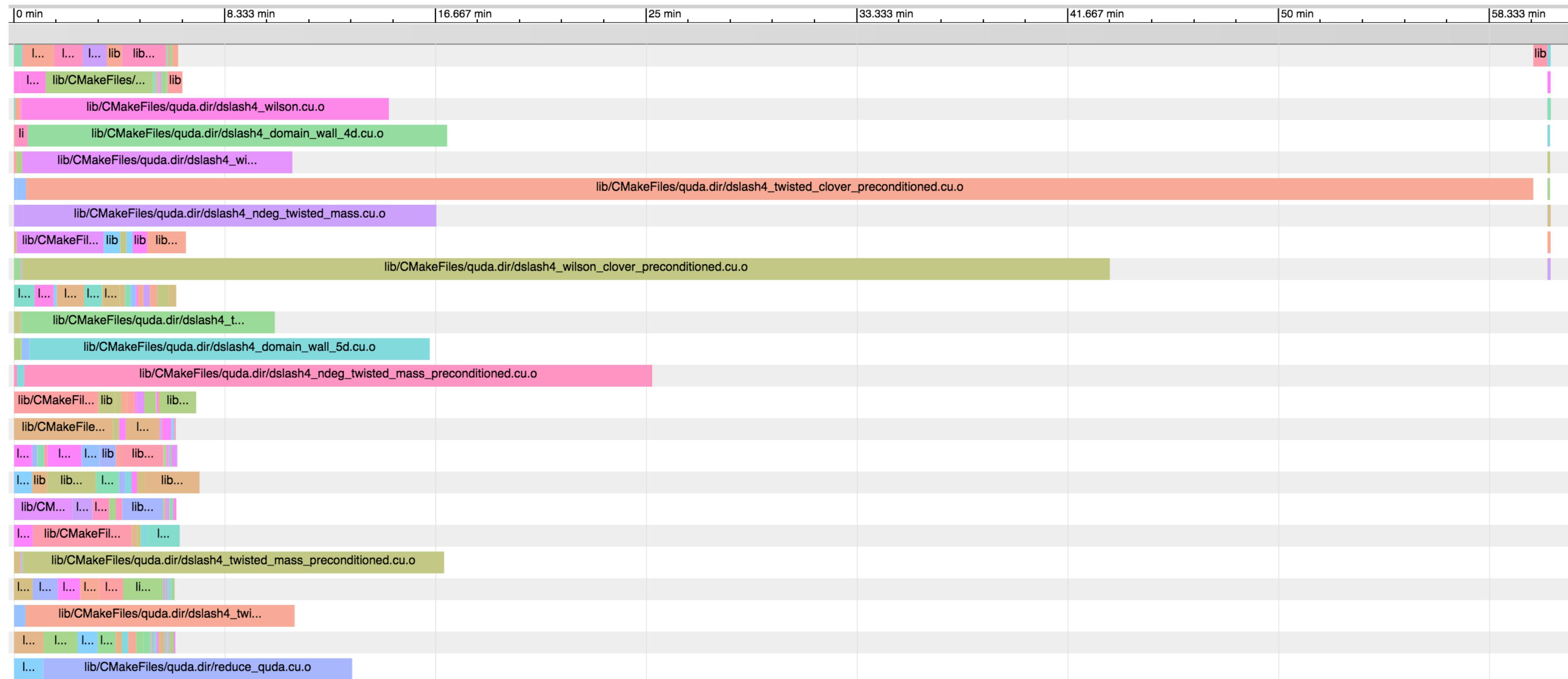
JIT: specify kernel name as string

Offline: static instantiation name

JIT: load kernel as string

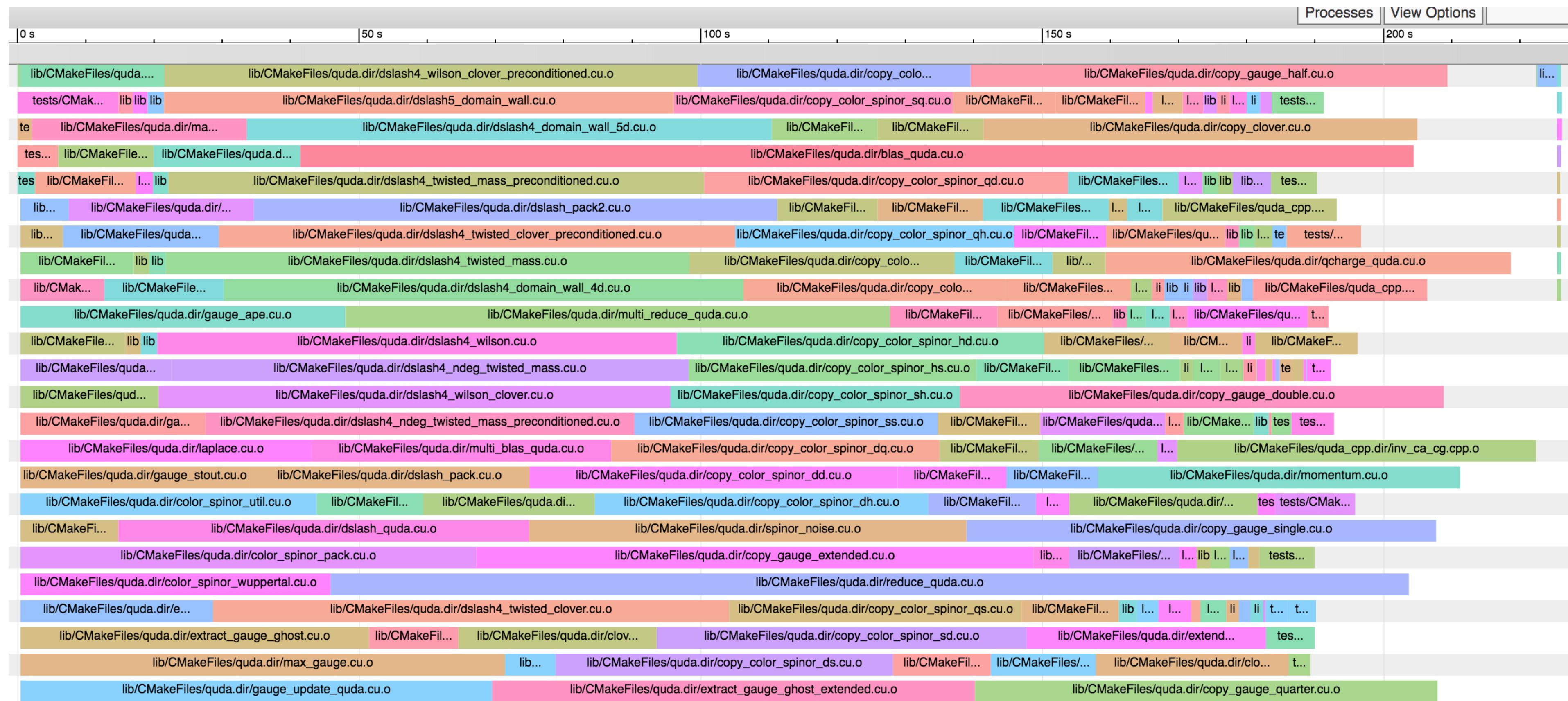
# COMPILATION COMPARISON

## Offline compilation



# COMPIILATION COMPARISON

## Just-in-time compilation



# ONGOING QUDA WORK

...or a subset of...

Multigrid

Continual improvements for clover (with Balint)

Staggered Multigrid (primarily Evan)

Reworked deflation and eigensolver framework (Dean)

Lanczos and Arnoldi

Deflation-accelerated multigrid (prolongator and coarse-grid)

Finish up ripping out old framework - QUDA will halve in size from 900K LOC -> 450K LOC

New stencils: Sextet fermions (with Ricky Wong), non-degenerate twisted-clover fermions, etc.

Internal @ NVIDIA: QUDA/LQCD is used for future architecture / compiler / driver developments

NVSHMEM (see Mathias' talk)

# QUDA ROADMAP

Release QUDA 1.0 (this summer)

Post 1.0

Multi-rhs block solvers for all stencils

Improved in strong scaling through NVSHMEM (see Mathias' talk)

Beyond just regular QCD

Longer term

Investigate how well QUDA runs on C++17 pSTL

Post feature requests here: <https://github.com/lattice/quda/issues>

# SUMMARY

The dslash kernel rewrite is the biggest change to QUDA in 10 years

- Will enable anyone to add support for new Dirac operators

- Enables next generation of algorithm techniques

Continual performance and algorithm improvements

BACK UP

# ACCESSORS

99% of QUDA is now written using “accessors”

Originally implemented for QUDA’s copy kernels to support application data layout

Opaque load and store functions that obfuscate data order

Data compression/decompression handled in the accessor

Gauge reconstruction

Fixed-point <-> floating point conversion and scaling

Trivial to add support for new data order

# MILC GAUGE ACCESSOR

```
template <typename Float, int length> struct MILCOrder : public LegacyOrder<Float,length> {
    typedef typename mapper<Float>::type RegType;
    Float *gauge;
    const int volumeCB;
    const int geometry;
    MILCOrder(const GaugeField &u, Float *gauge_=0, Float **ghost_=0) :
        LegacyOrder<Float,length>(u, ghost_), gauge(gauge_ ? gauge_ : (Float*)u.Gauge_p()),
        volumeCB(u.VolumeCB()), geometry(u.Geometry()) { ; }
    MILCOrder(const MILCOrder &order) : LegacyOrder<Float,length>(order),
        gauge(order.gauge), volumeCB(order.volumeCB), geometry(order.geometry)
    { ; }
    virtual ~MILCOrder() { ; }

    __device__ __host__ inline void load(RegType v[], int x, int dir, int parity) const {
        for (int i=0; i<length; i++) {
            v[i] = (RegType)gauge[((parity*volumeCB+x)*geometry + dir)*length + i];
        }
    }

    __device__ __host__ inline void save(const RegType v[], int x, int dir, int parity) {
        for (int i=0; i<length; i++) {
            gauge[((parity*volumeCB+x)*geometry + dir)*length + i] = (Float)v[i];
        }
    }

    __device__ __host__ inline gauge_wrapper<RegType,MILCOrder<Float,length> >
    operator()(int dim, int x_cb, int parity) {
        return gauge_wrapper<RegType,MILCOrder<Float,length> >(*this, dim, x_cb, parity);
    }

    __device__ __host__ inline const gauge_wrapper<RegType,MILCOrder<Float,length> >
    operator()(int dim, int x_cb, int parity) const {
        return gauge_wrapper<RegType,MILCOrder<Float,length> >
            (const_cast<MILCOrder<Float,length>&>(*this), dim, x_cb, parity);
    }

    size_t Bytes() const { return length * sizeof(Float); }
};
```

load accessor

save accessor

lhs wrapper ( `u(d,x,parity) = u;` )

rhs wrapper ( `u = u(d,x,parity);` )

```

template <typename Float, int Ns, int Nc>
struct SpaceColorSpinorOrder {
    typedef typename mapper<Float>::type RegType;
    static const int length = 2 * Ns * Nc;
    Float *field;
    size_t offset;
    Float *ghost[8];
    int volumeCB;
    int faceVolumeCB[4];
    int stride;
    int nParity;
SpaceColorSpinorOrder(const ColorSpinorField &a, int nFace=1, Float *field_=0, float *dummy=0, Float **ghost_=0)
: field(field_ ? field_ : (Float*)a.V()), offset(a.Bytes()/(2*sizeof(Float))),
volumeCB(a.VolumeCB()), stride(a.Stride()), nParity(a.SiteSubset())
{
    if (volumeCB != stride) errorQuda("Stride must equal volume for this field order");
    for (int i=0; i<4; i++) {
        ghost[2*i] = ghost_ ? ghost_[2*i] : 0;
        ghost[2*i+1] = ghost_ ? ghost_[2*i+1] : 0;
        faceVolumeCB[i] = a.SurfaceCB(i)*nFace;
    }
}
virtual ~SpaceColorSpinorOrder() { ; }

__device__ __host__ inline void load(RegType v[length], int x, int parity=0) const {
    for (int s=0; s<Ns; s++) {
        for (int c=0; c<Nc; c++) {
            for (int z=0; z<2; z++) {
                v[(s*Nc+c)*2+z] = field[parity*offset + ((x*Nc + c)*Ns + s)*2 + z];
            }
        }
    }
}

__device__ __host__ inline void save(const RegType v[length], int x, int parity=0) {
    for (int s=0; s<Ns; s++) {
        for (int c=0; c<Nc; c++) {
            for (int z=0; z<2; z++) {
                field[parity*offset + ((x*Nc + c)*Ns + s)*2 + z] = v[(s*Nc+c)*2+z];
            }
        }
    }
}

__device__ __host__ inline colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >
operator()(int x_cb, int parity) {
    return colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc>>(*this, x_cb, parity);
}

__device__ __host__ inline const colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >
operator()(int x_cb, int parity) const {
    return colorspinor_wrapper<RegType,SpaceColorSpinorOrder<Float,Ns,Nc> >(
        const_cast<SpaceColorSpinorOrder<Float,Ns,Nc>&>(*this), x_cb, parity);
}

```

### load accessor

### save accessor

# SPINOR ACCESSOR

```

__device__ __host__ inline void loadGhost(RegType v[length], int x, int dim,
                                         int dir, int parity=0) const {
    for (int s=0; s<Ns; s++) {
        for (int c=0; c<Nc; c++) {
            for (int z=0; z<2; z++) {
                v[(s*Nc+c)*2+z] =
                    ghost[2*dim+dir][(((parity*faceVolumeCB[dim]+x)*Nc + c)*Ns + s)*2 + z];
            }
        }
    }
}

__device__ __host__ inline void saveGhost(const RegType v[length], int x, int dim,
                                         int dir, int parity=0) {
    for (int s=0; s<Ns; s++) {
        for (int c=0; c<Nc; c++) {
            for (int z=0; z<2; z++) {
                ghost[2*dim+dir][(((parity*faceVolumeCB[dim]+x)*Nc + c)*Ns + s)*2 + z]
                    = v[(s*Nc+c)*2+z];
            }
        }
    }
}

size_t Bytes() const { return nParity * volumeCB * Nc * Ns * 2 * sizeof(Float); }

```

### loadGhost accessor

### saveGhost accessor

# WILSON-CLOVER KERNEL

```

template <typename Float, int nDim, int nColor, int nParity, bool dagger, KernelType
kernel_type, typename Arg>
__device__ __host__ inline void wilsonClover(Arg &arg, int idx, int parity)
{
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;
    typedef ColorSpinor<real,nColor,2> HalfVector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true;
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim,QUDA_4D_PC,kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;
    Vector out;
    // defined in dslash_wilson.cuh
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity,
idx, thread_dim, active);

    if (kernel_type == INTERIOR_KERNEL) {
        Vector x = arg.x(x_cb, my_spinor_parity);
        x.toRel(); // switch to chiral basis

        Vector tmp;

        #pragma unroll
        for (int chirality=0; chirality<2; chirality++) {
            HMatrix<real,nColor*Arg::nSpin/2> A = arg.A(x_cb, parity, chirality);
            HalfVector x_chi = A * x.chiral_project(chirality);
            tmp += x_chi.chiral_reconstruct(chirality);
        }

        tmp.toNonRel(); // switch back to non-chiral basis

        out = tmp + arg.kappa * out;
    } else if (active) {
        Vector x = arg.out(x_cb, my_spinor_parity);
        out = x + arg.kappa * out;
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
}

```

```

// CPU kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger,
KernelType kernel_type, typename Arg>
void wilsonCloverCPU(Arg arg)
{
    for (int parity= 0; parity < nParity; parity++) {
        // for full fields then set parity from loop else use arg setting
        parity = nParity == 2 ? parity : arg.parity;

        for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
            wilsonClover<Float,nDim,nColor,nParity,dagger,kernel_type>(arg, x_cb,
parity);
        } // 4-d volumeCB
    } // parity
}

// GPU Kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger,
KernelType kernel_type, typename Arg>
__global__ void wilsonCloverGPU(Arg arg)
{
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
        case 0: wilsonClover<Float,nDim,nColor,nParity,dagger,kernel_type>(arg, x_cb,
0); break;
        case 1: wilsonClover<Float,nDim,nColor,nParity,dagger,kernel_type>(arg, x_cb,
1); break;
    }
}

```

# WILSON-CLOVER PRECONDITIONED

```

template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
__device__ __host__ inline void wilsonClover(Arg &arg, int idx, int parity)
{
    using namespace linalg; // for Cholesky
    typedef typename mapper<Float>::type real;
    typedef ColorSpinor<real,nColor,4> Vector;
    typedef ColorSpinor<real,nColor,2> HalfVector;

    bool active = kernel_type == EXTERIOR_KERNEL_ALL ? false : true;
    int thread_dim; // which dimension is thread working on (fused kernel only)
    int coord[nDim];
    int x_cb = getCoords<nDim, QUDA_4D_PC, kernel_type>(coord, arg, idx, parity, thread_dim);

    const int my_spinor_parity = nParity == 2 ? parity : 0;

    Vector out;

    // defined in dslash_wilson.cuh
    applyWilson<Float,nDim,nColor,nParity,dagger,kernel_type>(out, arg, coord, x_cb, 0, parity,
    idx, thread_dim, active);

    if (kernel_type != INTERIOR_KERNEL && active) {
        // if we're not the interior kernel, then we must sum the partial
        Vector x = arg.out(x_cb, my_spinor_parity);
        out += x;
    }

    if (isComplete<kernel_type>(arg, coord) && active) {
        out.toRel(); // switch to chiral basis

        Vector tmp;

#pragma unroll
        for (int chirality=0; chirality<2; chirality++) {
            HMatrix<real,nColor*Arg::nSpin/2> A = arg.A(x_cb, parity, chirality);
            HalfVector out_chi = out.chiral_project(chirality);

            if (arg.dynamic_clover) {
                Cholesky<HMatrix,real,nColor*Arg::nSpin/2> cholesky(A);
                out_chi = static_cast<real>(0.25)*cholesky.backward(cholesky.forward(out_chi));
            } else {
                out_chi = A * out_chi;
            }

            tmp += out_chi.chiral_reconstruct(chirality);
        }

        tmp.toNonRel(); // switch back to non-chiral basis

        if (xpay) {
            Vector x = arg.x(x_cb, my_spinor_parity);
            out = x + arg.kappa * tmp;
        } else {
            out = tmp;
        }
    }

    if (kernel_type != EXTERIOR_KERNEL_ALL || active) arg.out(x_cb, my_spinor_parity) = out;
}

```

```

// CPU kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
void wilsonCloverCPU(Arg arg)
{

    for (int parity= 0; parity < nParity; parity++) {
        // for full fields then set parity from loop else use arg setting
        parity = nParity == 2 ? parity : arg.parity;

        for (int x_cb = 0; x_cb < arg.threads; x_cb++) { // 4-d volume
            wilsonClover<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, parity);
        } // 4-d volumeCB
    } // parity
}

// GPU Kernel for applying the Wilson operator to a vector
template <typename Float, int nDim, int nColor, int nParity, bool dagger, bool xpay, KernelType
kernel_type, typename Arg>
__global__ void wilsonCloverGPU(Arg arg)
{
    int x_cb = blockIdx.x*blockDim.x + threadIdx.x;
    if (x_cb >= arg.threads) return;

    // for full fields set parity from y thread index else use arg setting
    int parity = nParity == 2 ? blockDim.z*blockIdx.z + threadIdx.z : arg.parity;

    switch(parity) {
    case 0: wilsonClover<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 0); break;
    case 1: wilsonClover<Float,nDim,nColor,nParity,dagger,xpay,kernel_type>(arg, x_cb, 1); break;
    }
}

```

# DYNAMIC CLOVER

```
#pragma unroll
for (int chirality=0; chirality<2; chirality++) {

    HMatrix<real,nColor*Arg::nSpin/2> A = arg.A(x_cb, parity, chirality);
    HalfVector out_chi = out.chiral_project(chirality);

    if (arg.dynamic_clover) {
        Cholesky<HMatrix, real, nColor*Arg::nSpin/2> cholesky(A);
        out_chi = cholesky.backward(cholesky.forward(out_chi));
    } else {
        out_chi = A * out_chi;
    }

    tmp += out_chi.chiral_reconstruct(chirality);
}
```

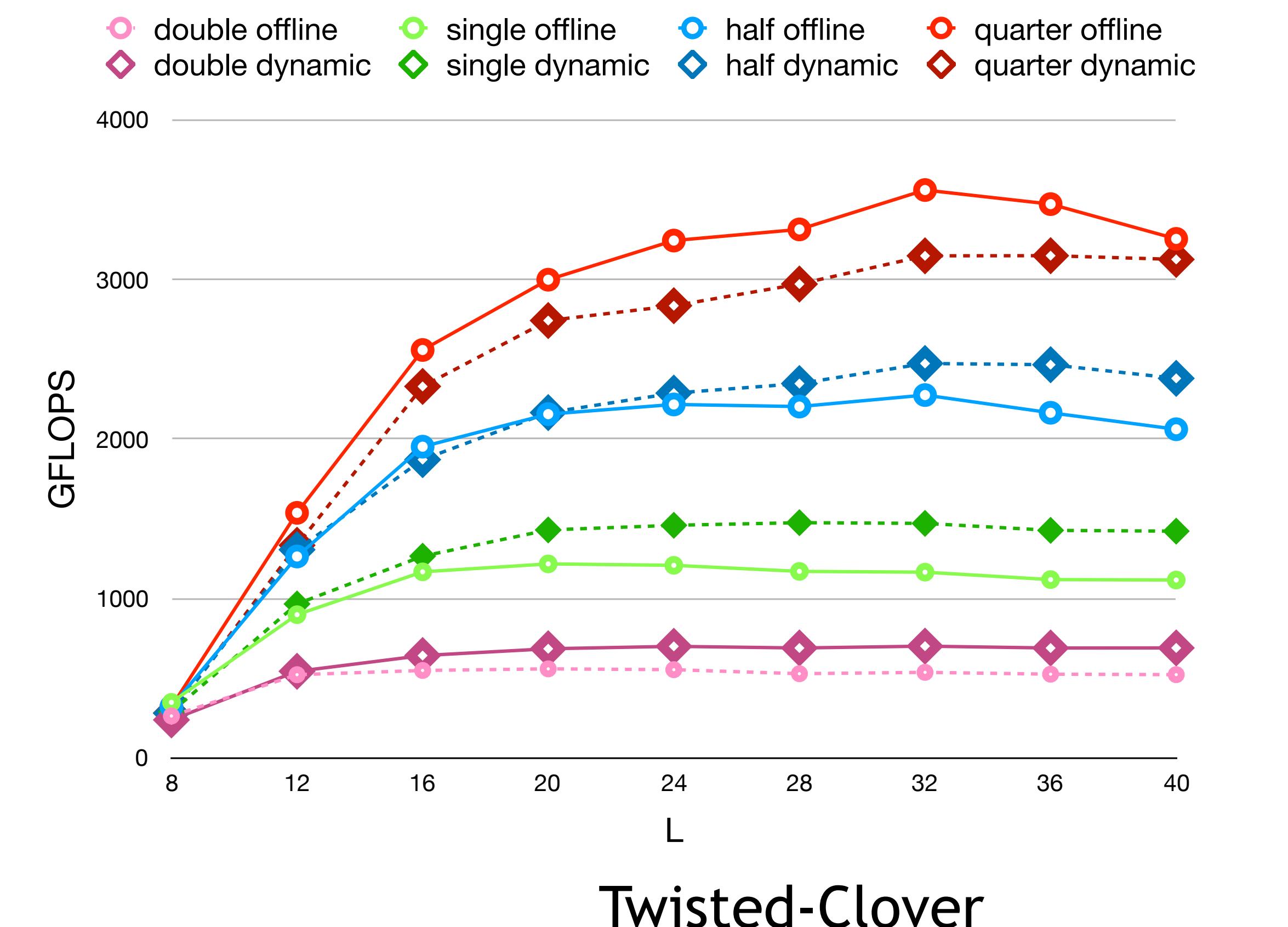
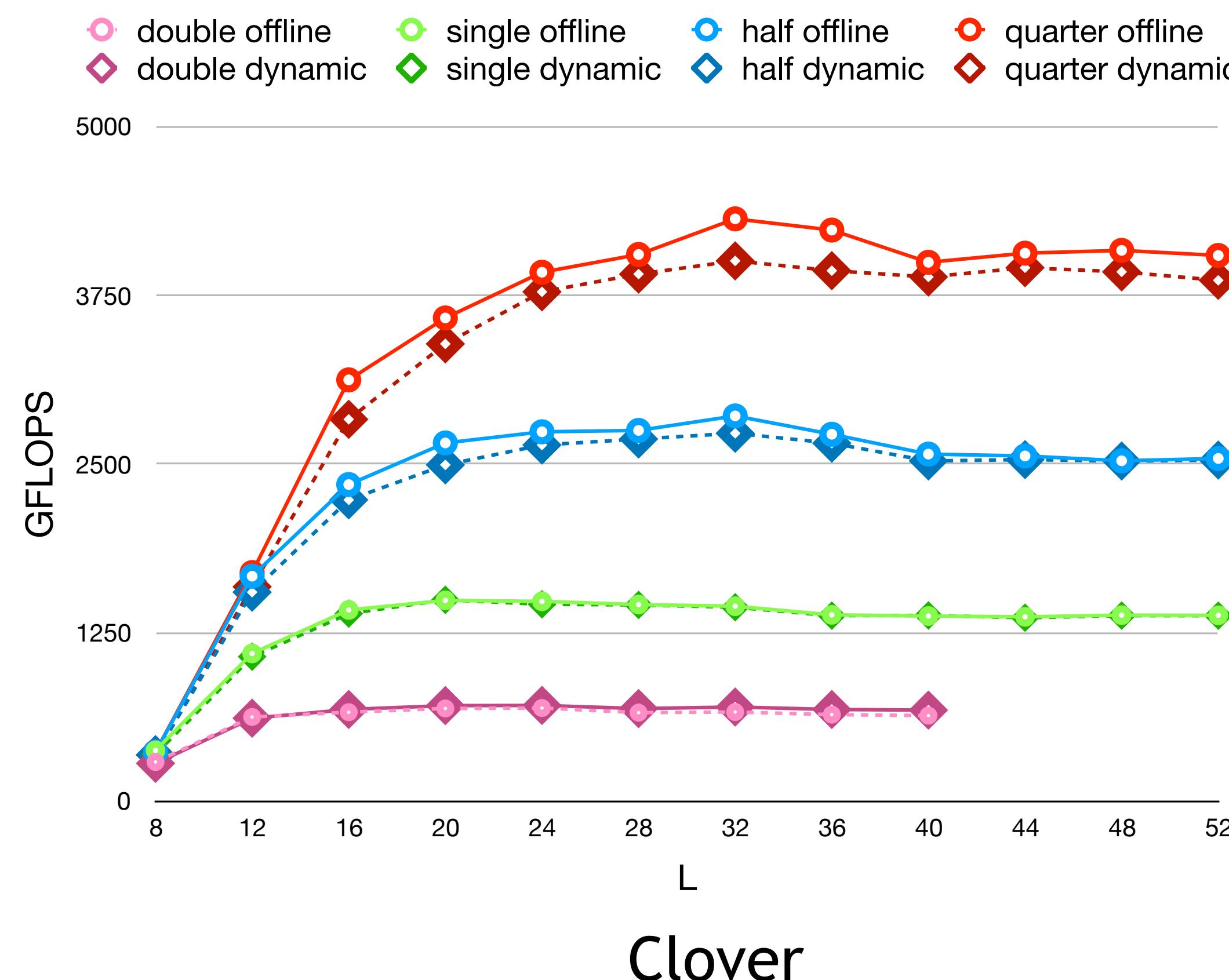
## Memory reduction strategy

No need to store both clover and inverse clover fields

Performance (and memory reduction) strategy

72 reals per site -> 54 reals per site (TODO)

# DYNAMIC CLOVER PERFORMANCE



```

template <typename Float, typename Arg>
class Stencil : public Tunable {

protected:
    Arg &arg;
    const Field &meta;

    long long flops() const
    {
        return 48*(long long)meta.VolumeCB();
    }
    long long bytes() const
    {
        return arg.out.Bytes() + 8*arg.in.Bytes();
    }
    bool tuneGridDim() const { return false; }
    unsigned int minThreads() const { return arg.volume; }

public:
    Stencil(Arg &arg, const Field &meta) : arg(arg), meta(meta)
    {
        strcpy(aux, meta_AUXString());
        strcat(aux, comm_dim_partitioned_string());
    }
    virtual ~Stencil() { }

    void apply(const cudaStream_t &stream) {
        TuneParam tp = tuneLaunch(*this, getTuning(), getVerbosity());
        stencilGPU<Float,nDim,nColor> <<<tp.grid, tp.block, tp.shared_bytes, stream>>>(arg);
    }
    TuneKey tuneKey() const { return TuneKey(meta.VolString(), typeid(*this).name(), aux); }
};

template <typename Float>
void ApplyStencil(Field &out, const Field &in)
{
    StencilArg<Float> arg(out, in);
    Stencil<Float, StencilArg<Float> > stencil(arg, in);
    stencil.apply(0);
}

```

## Performance metrics

## Tuning metadata

Launch work to CPU or GPU depending on field type

Unique “TuneKey” is for every kernel parameter set

1. Create launcher class instance
2. Launch work